Pascal
NEWSLETTER

NUMBER 10 OREGON SOFTWARE SPRING/SUMMER 1985

ULTRIX compiler joins the Pascal—2 family

The last six months have been an intense period of product development at Oregon Software.

Recent developments include the completion of Pascal—2 for VAX/ULTRIX-32; completion of the Oregon Linker/Assembler for our cross-development systems to the 68000; and completion of native SourceTools for VMS.

Once Pascal—2 for VMS was completed in the fall, we were able to quickly produce the ULTRIX compiler. Our technical team, led by Bruce Graham, used the compiler's common front end with the VAX/VMS code generator to produce a compiler that was retargeted for the ULTRIX system. With this cross-compiler, Bruce then compiled the M68000 UNIX support library, which is itself written in Pascal. This produced a Pascal support library for ULTRIX, which provided a testing environment for larger, more complicated programs. The ULTRIX product was put into field test in April and is being released to end users now.

Completion of the Oregon Linker/Assembler frees us from reliance upon the Oasys cross-linker/assembler,

VAX/VMS compiler retargeted for ULTRIX

which because of its slowness and incompatibility with standard M68000 object format was a weak link in our cross-tools packages from VMS and RSX. The Oregon Linker/Assembler runs under RSX and VMS native mode, making the cross-tools package available on all RSX and VMS configurations. In addition, our Linker/Assembler is designed to be largely machine-independent so that we can move it to new configurations without having to rely on other vendors' software in the future. In addition, the new versions of the cross-compiler itself (2.0P, shortly to be followed by 2.0Q) contain a number of bug fixes.

Native VMS SourceTools also completed field-testing and is being moved into regular distribution. Customers who purchased the RSX-to-VMS upgrade should be receiving their VMS versions shortly, if they haven't already.

We also completed the last round of bug fixes on Pascal—1, which is now an unsupported product. (See "Pascal—1 goes unsupported" later in this newsletter for update details.)

Long-term development

A major goal of Oregon Software has been to make our products — as well as our customers' — portable. Because most of our products are written in Pascal, we have largely succeeded. Our efforts to quickly move Pascal—2 to a variety of different machine architectures, however, have sometimes resulted in compromise. Over time, compiler sources have drifted apart as unique machine-oriented features and bug fixes have been added. For example, in order to make our compilers interface in a natural way with various operating systems, we have had to modify or rewrite command-line processing code for each machine. This has gradually increased maintenance and development problems.

Over the last few months, our compiler developers have been working on the "Common Front End" project. Their goal is to realign the source code for the compiler to provide a clear distinction between machine-dependent and machine-independent code.

The "front end" of the compiler is the code that scans Pascal source programs, analyzes syntax, and does the machine-independent optimizations. Th "back end" of the compiler is the code generator, which is unique to each machine. The code generators for our different products are now being interfa ed with the new common front end.

The common front end code served as the basis for the VAX/ULTRIX and the new 80186/286 project. Now that ou compilers have converged on a common set of sources, maintenance and develop ment will be much easier. Any new feature or bug fix in the front end code will immediately become available in each of the other compilers.

We began the 80186/286 project in April and now have a Pascal—2 compil

'Common front end' improves maintenance and development

for the IBM PC/AT running XENIX. V have a large memory model for code siz that takes advantage of the expanded instruction set of the 80286 processor and we are currently working on the co generation for 32-bit integer arithmetic. Our goal is to have an AT-based compil

Continued on Page .

In this issue . . .

Common front end fosters modularity

Oregon Software's technical staff has been hard at work the past few months. Perhaps most exciting has been the work done to consolidate the various Pascal—2 compilers into one uniform structure, isolating machine-dependent code and sharing source modules to a much larger extent than before.

This work is significant for two reasons. First, expanding the use of common source modules will speed bug fixing, increasing the reliability of our Pascal—2 products. A good example is our recent effort to make Pascal—2 comply 100 percent with the ISO standard (past versions have been very close, but not perfect). As a result of the shared source code, most of our Pascal—2 products can be made to comply at a fraction of the effort necessary in the past.

This work will also lessen the effort required to introduce new compiler products. For example, Oregon Software recently announced availability of Pascal—2 for VAX/ULTRIX. The VAX/VMS compiler and VAX/ULTRIX compiler differ in only two areas: the user interface (program/debug vs. program -debug), and the object/assembly code output routines for the code generator. We have truly modularized the compiler and have made heavy use of MAKE (a component of SourceTools) to manage the system. For instance, the VAX/ULTRIX project began with the implementation of a VAX/VMS to VAX/ULTRIX cross-compiler. After this compiler was completed, it only required slight modification of the MAKE file to replace VMS-host specific modules with ULTRIX-host specific modules to generate a VAX/ULTRIX-hosted native compiler.

Likewise, our Intel 80186/286 product (which has just entered field test) currently runs in native mode under PC-IX, as a cross-compiler under VAX/VMS, and as a cross-compiler under VAX/ULTRIX. These three versions are built by simply including the right pieces for each environment.

The net result of this effort? Better maintainability, as well as a quicker pace in development of new host-target environments supported by Pascal—2, which together mean better service to our customers.

Don R. Baccus, President

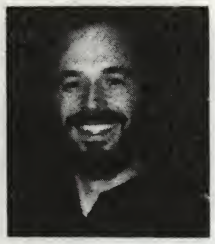
New editor named

We're changing editors again! Collins Hemingway began the *Oregon Software Pascal Newsletter*. In the fall of 1983, I took over from him (Number 7). With this issue, Tom Hanrahan becomes our newsletter's editor.

Tom has worked on our user manuals since February 1984 and, until recently, he was managing editor of the *Portland Family Calendar*, a very popular monthly newspaper. Tom has produced this issue in a new format and included an index to previous issues. He has some good ideas for forthcoming issues, too. But, one thing never changes: he'll need articles about your applications, your technical tricks and discoveries, your Pascal-related interests. We can't have a users' newsletter without submissions from users.

David B. Spencer

New employees draw on wide experience



Software engineer **Jan de Rie** joined Oregon Software in December. He comes to us from the Netherlands, where he worked for almost four years in Leiden. There, he co-introduced Pascal—2 in a large organization (more than 100 programmers) and implemented the Pascal—2 support library on the company's proprietary operating system. Jan attended Erasmus University in Rotterdam, where he earned an M.A. in mathematical economics. Jan's main interest at Oregon Software is working with the Pascal compilers. Currently, he is developing a version of the Pascal—2 compiler for the Intel 80186/286 family of processors. Jan actively pursues his hobby of folkdancing and performs with a dance group around the northwest area.

As our newest account executive, **Ruby Haughton** will soon be introducing herself to many of our customers. Ruby came to Oregon Software in May from the sales department at Proactive, a division of Pacific Telecom Co. A graduate from the University of Oregon, Ruby brings with her the skills and experience needed to provide new prospects with technical product descriptions and to respond to our current customers' requests for help and additional product information. Outside of Oregon Software, Ruby works with adolescents as a volunteer counselor, and she enjoys singing. She'll be touring the Caribbean this summer as a counselor/performer of an international sports evangelical team.

Steve Bihler joined Oregon Software in mid-January as Technical Support Manager. His primary function will be assisting customers with any technical problems they may encounter. Steve's previous job was as a Senior Systems Engineer at Perkin-Elmer's Technical Support Center. He has an interdisciplinary degree from Washington State University in computer science and business administration. He likes flying, skiing, and radio communications. Steve has lived in more than 24 different places around the world including Greece and the Azores, and he has traveled extensively through Europe and the Orient.

OPUS Communiqué

Oregon Pascal Users Society

Bruce Williams, who served as the OPUS coordinator since the group's formation in November 1982, has given up his position as head of the Society. Any OPUS member interested in taking over as the Society's coordinator should contact:

Tim McMenamin
Oregon Software
6915 SW Macadam Ave.
Portland, OR 97219
(503)245-2202

The principal duties of the OPUS coordinator are to:

- Collect information submitted by OPUS members and disseminate it through this column.
- Facilitate contact among OPUS members by matching one member's problems with another member's expertise.
- Maintain and coordinate the OPUS shared library.

Over the past few years, OPUS has accomplished many of its initial goals. Membership has grown to over 100 individuals from around the world. The Society has helped numerous users of Pascal—1 and Pascal—2 with programming problems—everything from simple character I/O to asynchronous-trap routines, and most recently, the Society has taken steps to establish a shared library of routines and utility programs.

But clearly, the continued success of the Oregon Pascal User Society depends on the efforts of its membership and its ability to draw upon its membership to fill key positions within the organization.

Pascal—1 goes unsupported

Pascal—1 version 1.3D was released for field test on April 8, 1985. Version 1.3D is the final release of our Pascal—1 compiler. The software is stable in its present form, and Oregon Software plans no subsequent releases.

Pascal—1 is Oregon Software's first Pascal compiler. Written entirely in MACRO-11, the compiler runs on DEC's PDP-11 series machines. It implements Pascal in much the same form as originally conceived by Jensen and Wirth with some extensions, such as separate compilation facilities.

Pascal—1 has been in commercial use since 1977. Thanks to its quick compilation and its minimal memory demands, Pascal—1 was rapidly embraced by the academic community, where it still enjoys wide use.

In the interest of portability, improved code optimization, and adherence to the international Pascal standard (ISO 7185), Oregon Software troduced its new compiler, Pascal—2, summer 1981. As Pascal—2 far exceed Pascal—1 in performance and portabi we propose to devote our resources to maintaining and enhancing the newer compiler.

Oregon Software will no longer s support contracts for Pascal—1. Customers who were under support fc Pascal—1 as of January 1984 are eligi to receive the final update free of cha and should contact our customer serv department to request their upgrade. Customers who are not eligible for th free update and wish to purchase the software should call our sales staff fo price information and product availability.

Pascal
NEWSLETTER
OREGON SOFTWARE
6915 SW Macadam Avenue
Portland, Oregon 97219

Thomas E. Hanrahan, editor
David Spencer, David Barnes, Collins Hemingway, writers
Betsy Slonaker, production assistant

The Pascal Newsletter is published regularly by Oregon Software, Inc., 6915 SW Macadam Ave., Portland, OR 97219; (503) 245-2202. Each customer of Oregon Software receives one free subscription per site. Additional subscriptions are available upon written request.

The Pascal Newsletter accepts articles of interest to Pascal users: solutions to troublesome programming situations, new applications of Pascal, interesting variations on standard applications, etc. Submit articles on paper (typed and double-spaced), on floppy disk, or on magnetic tape, sent to the attention of the editor. We pay $100 per newsletter page for any article we print.

Printed in USA

RSX virtual memory window speeds record access

y Steve Poulsen

RSX system services known as ıemory management directives control ıe way in which memory management ardware of a PDP-11 computer maps a rogram's virtual memory address space ɔ physical memory. In general, Pascal rograms running on a mapped RSX ystem have access to a maximum of 64K ytes of memory. The limitation is imosed by the 16-bit nature of PDP-11 omputers — at any instant, a program ıay not have direct access to more than 4K bytes of virtual address space. But hysical memory on most PDP-11 comuter considerably larger than 64K ytes and as long as you do not attempt ɔ directly reference more than 64K bytes t a time, you can write Pascal programs ıat call upon the memory management irectives to take advantage of additional ıemory resources.

The physical memory accessible to a ısk is called a region. (Shared libraries nd shared common areas are examples f regions.) The block of physical ıemory in which a task runs is called the task region." Access to regions is ırough "windows," which are a portion f a task's virtual memory. Each window ıust start at an address that is a multiple f 4K words. So, in any 32K word task,

Programs can access additional memory on most PDP-11s

ıp to 8 windows may be defined. The ize of a window may vary from 32 words o 32K words. If a region is larger than he size of a window, the window may be noved to different locations within the egion. For example, a 4K word window nay be mapped to many different ›hysical addresses within a 100K word egion. The physical size of a region is imited only by the size of the partition n which the region resides.

To use virtual memory, a Pascal proɡram must call upon the RSX memory nanagement directives to:

- Attach to an exiting or dynamically created region.
- Create an address window to use in accessing a region.
- Map a window into a region.

The RSX memory management directives (described in the *RSX-11M/PLUS Executive Reference Manual*) may be called directly by a Pascal progam through FORTRAN entry points in your system library (LB:[1,1]SYSLIB.OLB). To provide an interface between a Pascal program and the RSX directives, you must create two special data structures: a region definition block and a window definition block, both of which may be represented as simple Pascal records. The fields for these two structures correspond to the block parameters defined for specific memory management directives described in the executive reference manual.

The sample program VIRT.PAS, presented on the following page, is an example of how to call RSX memory management directives from a Pascal program. Explanations within the text of the code appear as ordinary Pascal comments. As the example shows, the program contains Pascal record definitions for the region definition and window definition blocks used by the system services. In addition, VIRT.PAS contains the nonpascal procedure declarations required in calls to the memory management directives.

The subroutines in VIRT.PAS implement a simple virtual file system by creating a dynamic region and then reading and writing data records within the region in much the same way that a disk file operates. The VINIT procedure initializes the virtual memory system by creating a dynamic region called PASDAT in the GEN partition. The user specifies the size of the region. VINIT also creates an address window that maps APR7 into the dynamic region. This causes virtual addresses in the range 160000 to 177777 to be mapped into the dynamic region.

To use VIRT.PAS, you must first modify the code to define the type of data to be stored in the virtual file. An integer is used for demonstration purposes in the current example. Access to the records in virtual memory is through the procedures VSEEK, VGET, and VPUT. These procedures each use a record number to compute the physical location of the desired record in the dynamic region. Then, if necessary, a window is mapped into the region so that the desired record is contained within the window. This makes the record available to the task. The data can be read, updated in place, or copied to some other record in the task. In its current form, the VIRT program must be compiled

Virtual file systems are very efficient

separately from its calling program because it uses OWN storage to maintain the current mapping information. The routine can easily be adapted for use as an include module by removing the $own directive.

The Task Builder cannot predict how many windows your program might create. So, when you task-build a program that uses virtual memory, you must instruct the Task Builder to build additional window data structures. The WNDWS = *n* option creates additional window blocks in the task's header. For example, if your program is called TEST, you could task-build it with the following commands:

```
> TKB
TKB > TEST/FP/CP = TEST,VIRT,
TKB > LB:[1,1]PASLIB/LB
TKB > /
ENTER OPTIONS:
TK > WNDWS = 1
TKB > //
```

When you create windows, you should base them at high virtual addresses. In the current example, APR 7 (the highest available APR) is used for the mapping. If your task maps to a resident library such as PASRES, you may need to use a tower APR for the window because Pascal tasks extend the size of the task (window 0) when additional heap space is required. When you are also using virtual memory for data or for virtual overlays, it is possible that the program may accidentally extend into virtual address already being used for other purposes.

```
PROGRAM virt;

CONST
    apr = 7; { APR to use for mapping }
    blockettes_per_4kw = 128; { 8192 div 64 }

TYPE
    data = integer; { Simple test case }
    data_pointer = ^data; { Pointer to data record }
    byte = 0..255; { Unsigned byte }
    word = 0..65535; { Unsigned word }
    name = PACKED ARRAY [1..6] OF char; { Name to convert to rad50 }
    rad50_name = ARRAY [1..2] OF word; { 6 rad50 chars }
    directive_status = word; { Directive status returned from exec call }
    region_id = word; { System supplied region ID }
    region_status_bits = (rs$red, rs$wrt, rs$ext, rs$del, rs$nex, rs$att,
                          rs$ndl, rs$mdl, rs$xx1, rs$xx2, rs$xx3, rs$xx4,
                          rs$xx5, rs$xx6, rs$unm, rs$crr);
    region_status = PACKED SET OF region_status_bits;
    access_bits = (pread, pwrite, extend, delete);
    access_category = (system, owner, group, world);
    accesses = PACKED SET OF access_bits;
    protection_word = PACKED ARRAY [access_category] OF accesses;

    region_block =
      RECORD
        gid: region_id; { System supplied region id }
        gsiz: word; { Size of region in blockettes [64 bytes each] }
        gnam: rad50_name; { Region name [or zeroes] }
        gpar: rad50_name; { Partition in which region resides }
        gsts: region_status; { Region control & status bits }
        gpro: protection_word; { Region protection word }
    END;

    window_status_bits = (ws$red, ws$wrt, ws$ext, ws$del, ws$bps, ws$sis,
                          ws$rcx, ws$map, ws$64b, ws$nat, ws$res, ws$nbp,
                          ws$rrf, ws$elw, ws$unm, ws$crw);
    window_status = PACKED SET OF window_status_bits;

    window_block =
  PACKED RECORD
    nid: byte; { System supplied window id }
    napr: byte; { APR to use to map virtual addresses }
    nbas: word; { Virtual base address of window [APR * 8192] }
    nsiz: word; { Size of window }
    nrid: region_id; { ID of region which this window maps into }
    noff: word; { Offset [in blockettes] into region }
    nlen: word; { Length [in blockettes] to map [0 = default] }
    nsts: window_status; { Window control & status bits }
    nsrb: word; { Send/receive buffer address [not used] }
END;

VAR
    data_size: word; { Size in bytes of DATA }
    max_record: word; { Max record number }
    records_per_4kw: word; { Number of data records in each 4KW block }
    current_4kw_block: word; { Block currently mapped }
    rdb: region_block; { Region definition block }
    wdb: window_block; { Window definition block }
```

```
PROCEDURE atrg(VAR rdb: region_block;
               VAR status: directive_status);
  NONPASCAL; { Attach region }

PROCEDURE crrg(VAR rdb: region_block;
               VAR status: directive_status);
  NONPASCAL; { Create region }

PROCEDURE dtrg(VAR rdb: region_block;
               VAR status: directive_status);
  NONPASCAL; { Detach region }

PROCEDURE craw(VAR wdb: window_block;
               VAR status: directive_status);
  NONPASCAL; { Create address window }

PROCEDURE elaw(VAR wdb: window_block;
               VAR status: directive_status);
  NONPASCAL; { Eliminate address window }

PROCEDURE map(VAR wdb: window_block;
              VAR status: directive_status);
  NONPASCAL; { Map address window }

PROCEDURE umap(VAR wdb: window_block;
               VAR status: directive_status);
  NONPASCAL; { Unmap address window }

PROCEDURE cvt_rad50(n: name;
                    VAR r: rad50_name);

  FUNCTION rad50(a, b, c: char): word;

    FUNCTION rad(c: char): word;

      BEGIN { Rad }
        IF c = ' ' THEN rad := 0
        ELSE IF c IN ['A'..'Z'] THEN rad := ord(c)—ord('A') + 1
        ELSE IF c IN ['0'..'9'] THEN rad := ord(c)—ord('0') + 30
        ELSE IF c = '$' THEN rad := 27
        ELSE IF c = '.' THEN rad := 28
        ELSE IF c = '_' THEN rad := 29
        ELSE
          BEGIN
          writeln('*', c, '* is not a legal RAD50 character!');
          rad := 0;
          END;
    END; { Rad }

  BEGIN { Rad50 }
    rad50 := (rad(a) * 40 + rad(b)) * 40 + rad(c);
  END; { Rad50 }

BEGIN { Cvt_Rad50 }
  n[1] := rad50(n[1], n[2], n[3]);
  n[2] := rad50(n[4], n[5], n[6]);
END; { Cvt_Rad50 }
```

The program VIRT.PAS provides an example of how to use virtual memory from Pascal. The subroutines in VIRT.PAS implement a virtual file system that is more efficient and has faster accessibility than a similar file system maintained on disk.

```
PROCEDURE vinit(number_of_records: word);
  EXTERNAL;

PROCEDURE vinit;

  VAR
    n_4kw_blocks: word; { Number of 4KW blocks needed to hold virtual data }
    remainder: word; { Blockettes required in final portion }
    status: directive_status; { Directive status }

  BEGIN { Vinit }
    data_size := size[data];
    IF odd[data_size] THEN data_size := data_size + 1;
    max_record := number_of_records;
    records_per_4kw := 8192 DIV data_size;
    n_4kw_blocks := number_of_records DIV records_per_4kw;
    remainder := ((number_of_records MOD records_per_4kw) * data_size +
                  63) DIV 64;

    WITH rdb DO
      BEGIN
      gid := 0;
      gsiz := n_4kw_blocks * blockettes_per_4kw + remainder;
      cvt_rad50('PASDAT', gnam); { Region name }
      cvt_rad50('GEN ', gpar); { Partition containing region }
      gsts := [rs$att, rs$wrt, rs$red, rs$mdl];
      gpro[system] := [ ];
      gpro[owner] := [ ];
      gpro[group] := [delete, extend, pwrite];
      gpro[world] := [delete, extend, pwrite, pread];
      END;
    crrg[rdb, status];
    IF status < > 1 THEN
      writeln('Can''t create region. Status = ', status: 1, ' . ');
    WITH wdb DO
      BEGIN
      nid := 0;
      napr := apr;
      nbas := 0;
      nsiz := blockettes_per_4kw;
      nrid := rdb.gid;
      noff := 0;
      nlen := 0;
      nsts := [ws$wrt];
      nsrb := 0;
      END;
    crew(wdb, status);
    IF status < > 1 THEN
      writeln('Can''t create address window. Status = ', status: 1, ' . ');
    current_4kw_block := -1;
  END; { Vinit }

FUNCTION vseek(record_number: word): data_pointer;
  EXTERNAL;

FUNCTION vseek;

  VAR
    needed_4kw_block: word; { Block containing desired record }
    offset_in_4kw_block: word; { Position of record in block }
    status: directive_status; { Status of MAP directive }

  BEGIN { Vseek }
    record_number := record_number-1; { Make relative to zero }
    IF (record_number < 0) OR (record_number > max_record) THEN
      BEGIN
      writeln('Virtual record number ', record_number + 1: 1,
              ' is out of range');
      needed_4kw_block := 0;
      offset_in_4kw_block := 0;
      END
    ELSE
      BEGIN
      needed_4kw_block := record_number DIV records_per_4kw;
      offset_in_4kw_block := (record_number MOD records_per_4kw) *
                                 data_size;
      END;
    IF needed_4kw_block < > current_4kw_block THEN
      BEGIN { Map new block containing desired record }
      wdb.noff := needed_4kw_block * blockettes_per_4kw;
      wdb.nlen := 0;
      map(wdb, status);
      IF status < > 1 THEN
        writeln('Can''t map to record ', record_number: 1, '. Status = ',
              status: 1, ' . ')
        ELSE current_4kw_block := needed_4kw_block;
        END;
    vseek := loophole(data_pointer, wdb.nbas + offset_in_4kw_block);
  END; { Vseek }

PROCEDURE vget(record_number: word;
                    VAR d: data);

  EXTERNAL;

PROCEDURE vget;

  VAR
    p: data_pointer;

  BEGIN { Vget }
    p := vseek(record_number);
    d := p^;
  END; { Vget }

PROCEDURE vput(record_number: word;
                    VAR d: data);

  EXTERNAL;
PROCEDURE vput;

  VAR p: data_pointer;

  BEGIN { Vput }
    p := vseek(record_number);
    p^ := d;
  END; { Vput }
```

You can prevent Pascal tasks from using additional virtual memory by means of a global patch. The global variable P$NEXT can be patched to contain a value of 240 to prevent Pascal from extending into additional virtual addresses, as shown:

```
> TKB
TKB > TEST/FP/CP = TEST,VIRT,
TKB > LB:[1,1]PASLIB/LB
TKB > /
ENTER OPTIONS:
TKB > GBLPAT = TEST:P$NEXT:240
TKB > EXTSCT = $$HEAP:20000
TKB > //
```

In earlier versions of Pascal this symbol is named $NOEXT.

When you prevent Pascal from using additional memory, you must provide explicit dynamic memory for the program by extending the heap via the EXTSCT option as shown in the previous task-build example. For each program, the amount of memory required for heap storage varies according to the amount of local variable storage and the number of files your program opens. If your program dies with a stack overflow or with a not enough memory error, you should increase the size of the heap.

With the patch just described, your Pascal program will never use any additional virtual memory. This gives you complete control over the allocation of virtual memory within the task.

One of the principal advantages gained by using virtual memory is speed. In the case of the virtual file system described here, records are read and written by copying the data from the task's memory into the memory of a dynamic region. Records are accessed by moving a region's mapping window instead of by reading a disk file. This memory-to-memory copy operation is very efficient and can be done in a matter of microseconds rather than milli-seconds.

Steve Poulsen is currently manager of Oregon Software's technical sales support. As one of the founders of Oregon Software, Steve has contributed heavily over the years to the company's development effort.

Rites of Spring . . .

Party Thrives in New Setting

Oregon Software's annual corporate party on May 17 moved indoors for the first time this year. Previously, we had always celebrated the company's incorporation at our old building on Canyon Road, where ample grounds and a secluded street made the commemoration a unique and well-attended event within the community. There were skeptics among those attending past celebrations who believed that the flavor of the old days would be lost in our new facility.

Like a die-hard C programmer being explained the benefits of Pascal, the skeptics were certain that such a conversion could never be made. What was going to happen to the bountiful barrels of imported beer that graced the halls of the previous parties? Probably replaced by wine coolers. What about the live folk music of the olden days? Probably done in by piped-in music from the dentist office down the street. Doubters were quickly reassured, however, as the party's featured event, Oregon Software's first-ever "Chili Cook-Off," immediately set the tone for much of the evening's activities.

An entire validation suite of tests was run on those entries submitted. The judges savored each dish and carefully cleansed their palates with Dos XX beer before moving on to the next entry. The dishes were tested for hotness, greasiness, viscosity, taste, and originality. After a long discussion, the judges decided that Bob Phillips, scoring 94 out of a possible 100 points, had won. The competition was extremely close, but Bob's chili had the robustness of a Pascal—2 compiler, and that in the end made the difference.

Competition among the finalists in Oregon Software's first Chili Cook-Off was close, but Bob Phillips's "Rump-p entry was named the contest's winner.

First prize was a night for two at Neskowin Resort on the Oregon coast.

After the judging, party-goers wei invited to decide for themselves which the chilies they preferred, as they dran margaritas and listened to mandolin fc music performed by Sandy Profeta an Jan DeWeese.

RSTS/E Version 9 User's Note

We recently received our copy of the latest RSTS/E release from Digitial Equipment Co. and have begun the process of testing and converting Pascal—2 to run under Version 9 of the operating system. We will announce the release of any new software required to run on Version 9 as soon as it is ready for distrib tion. Customers with current support contracts who are upgrading to Versio can request the new release of Pascal– as soon as our announcement of its availability is made.

Publication list grows as product list expands

by David Spencer

When I started at Oregon Software a little over three years ago, we had three technical writers, three existing manuals, and one new manual in progress. Today, we have three writers, seventeen existing manuals, and a new one in progress. Maintaining and improving our manuals demands more of our resources than producing new documentation.

A case in point: About two years ago, we announced that new editions of the Pascal—1 user manuals for the PDP-11 products were forthcoming (*Pascal Newsletter* Number 6, Spring 1983). What started out as a six-month project took a lot longer. While working on the PDP-11 books, we created manuals for Pascal—2 on three new microprocessors, for SourceTools on three DEC operating systems, and for the multi-optioned cross-development system on two hosts. With the publication of the Pascal—1 V1.3 manuals in April, we've fulfilled our two-year-old promise and completed basic documentation on all current products.

Despite staff turnover and size limitations, the Technical Publications group plans to improve and extend those manuals as part of its routine maintenance. New sections on the support library, on floating-point format, and our implementation of Pascal are already in progress. We're experimenting with larger typefaces and more white space in our page layouts. Late in the summer, we will begin another cycle of comprehensive revisions.

As we plan the next round of upgrades, we're looking through our files of users' suggestions and the feedback from our distributors' workshops. We're coordinating our efforts with the newly formed support group to gather further indications of your needs. We're developing better methods of production and graphic presentation, in order to make our manuals easier to read and use. We need your feedback on our manuals to continue improving our product documentation. This is an opportune time to send us that suggestion you've always thought you'd make.

OREGON SOFTWARE USER MANUALS: When we last printed a list of manuals and prices in this newsletter, we listed only Pascal—1, Pascal—2, SourceTools, Sort-1-Plus, and the concurrent package. We are again publishing a complete list of user manuals available. Prices and ordering information are provided in the Oregon Software Catalog of Products and Price Schedule *(February 1985).*

Oregon Software User Manuals

Title	Publication Date
Pascal—2 V2.1 for RSX	July. 83*
Pascal—2 V2.1 for RSTS	Aug. 83*
Pascal—2 V2.1 for RT-11	Aug. 83*
Pascal—2 V2.1 for VMS	Oct. 84*
Pascal—2 V2.1 for VAX/UNIX	May 85
Pascal—2 V2.1 for UNIX/68000	Aug. 83
Pascal—2 V2.1 for RT-11/Pro300	Feb. 85
Pascal—2 V2.1 for POS/Pro300	Feb. 85**
SourceTools V1.0 for VMS, RSX, RSTS	Apr. 85
Pascal—2 V2.0 for VERSAdos	Nov. 82*
Pascal—2 V2.0 Cross-Development RSX, VMS	Mar. 84*
Oregon Linker/Assembler RSX, VMS	May 85
Pascal—2 Concurrent Programming Pkg.	Apr. 84*
Pascal—1 V1.3 for RSX	Mar. 85
Pascal—1 V1.3 for RSTS	Mar. 85
Pascal—1 V1.3 for RT-11	Mar. 85
Sort-1-Plus V1.7	Apr. 80

* Items marked with an asterisk are scheduled for an update within the next three months.

** Abbreviated manual.

Pascal—2 compiler available on ULTRIX

Oregon Software's Pascal—2 compiler now runs on ULTRIX-32, Digital's version of the UNIX operating system for VAX and MicroVAX computers, and on Berkeley UNIX (BSD4.2), from which ULTRIX is derived.

As with all Pascal—2 releases on Digital systems, Pascal—2 on ULTRIX-32 includes a high-level interactive debugger, an execution profiler and other utilities that aid in coding and development.

Pascal—2 conforms to the international Pascal standard (ISO 7185) and produces extremely concise, fast object code, as compared to that produced by languages such as FORTRAN-77 and C. Pascal—2 supports all features of standard Pascal, including packed data structures and set types of up to 256 elements. The compiler allows calls to separately compiled routines written in Pascal, C, FORTRAN or MACRO, giving developers access to existing software libraries. Compile-time error-checking ensures data type conformance to the Pascal standard, locates many uninitialized variables and detects nearly 150 Pascal syntax errors.

ULTRIX-32 is a licensed version of Berkeley UNIX that supports the entire BSD4.2 command set, including virtual memory demand paging. Different versions of ULTRIX-32 run on MicroVAX and VAX-11/700 Series computers.

Pascal—2 for VAX/ULTRIX is now available from Oregon Software or its authorized distributors.

Book Review

by David M. Barnes

David M. Chess, *Programming in IBM PC DOS Pascal*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632, 1985.

Programming in IBM PC DOS Pascal is written for the Pascal programmer developing applications on the IBM PC. The language described is PC DOS Pascal, versions 1.0 and 2.0, which is sometimes referred to in the text as "Pascal 2," by the way. Don't be fooled !

Because Pascal is an excellent teaching language, many Pascal books take the form of programming tutorials. In *Programming in IBM PC DOS Pascal*, the reader is referred to a standard introductory Pascal text for language details. This allows the author, who seems to be in touch with the needs of the applications programmer, to concentrate on the extensions of this dialect of Pascal and its interactions with the PC DOS operating system.

The book is divided into three parts. Part One is "Essential Pascal." It deals with the language itself, introducing the extensions peculiar to this dialect. The author accurately describes this part of the book as a "whirlwind tour." Part Two is "The Details," about the same length as part one, and deals with data files, screen handling, keyboard I/O and other DOS services, such as the assembler interface. This is the main part of the book. While it may slight some programmers' pet topics, it covers many important things that a developer needs to know. Part Three is built around a small programming project that ties in some of the earlier examples in the book. Mr. Chess also touches on some of the mandatory Pascal subjects: style, modularity, and top-down design. On these subjects he is refreshingly un-dogmatic.

PC DOS Pascal has many extensions to Standard Pascal, such as conditional compilation, loop terminators, separate compilation facilities (including a Modula-2 look-alike called a "unit"), an optional pseudo-code listing, and a variety of additional data types. Also provided are built-in functions that access PC DOS system services.

Differences between the two versions are noted where they're important. Also noted, most of the time, are deviations from standard Pascal. Because the language's extensions are in large part the subject of the book, many deviations go unremarked. The author spends some time educating the reader in the uses of DOS, as well as Pascal, though without dissecting the operating system on an assembler level. Little time is spent on running the compiler (presumably covered in the manual) and the linker (also covered elsewhere).

The presentation of the book is good. It's slim, can be made to lie flat. Pascal keywords occurring in the text are printed in typewriter font so they can be picked out quickly when scanning. There is a summary at the top of each chapter, and the author encourages the reader to skip chapters in which the material seems too familiar.

Since all of the information exists in other forms, the strength of the book lies in how well its author anticipates the interests of his readers. I think he's done a remarkable job. The emphasis is on pragmatic information. The "Assembler Interface" chapter, for example, does not attempt to teach the reader PC assembly language, but it does provide the simplest possible assembler routine callable from Pascal.

The book's weakness is that it is (necessarily) sketchy in some areas that may be of interest to many programmers, and the level of understanding at which the book aims doesn't seem consistent throughout. There is sometimes a remarkable lack of precision in order to accommodate the novice reader, as in this description of `char` type variables:

"Char variables ... take values which are letters, punctuation marks, digits, spaces, and in general the sort of thing that one writes down."

The screen handling chapter is a good example of the author's approach. Mr. Chess sets forth some reasons for devoting attention to the manner in whic a program displays its output, then laun ches into describing four methods for doing so. These are: standard Pascal readln and writeln statements, ANSI.SY calls, BIOS calls, and writing directly to screen memory. Each section is illustrate with specific examples. Finally, he offer "Some Speed Considerations," in which he mentions the tradeoffs involved in selecting one method over another. The quantity of technical material presented surprising, especially considering that tl chapter begins with the sentence "The display screen is the PC's voice."

The goal of the book, though the author didn't state it this way, seems to to get the Pascal applications programn up and running on his IBM PC as quicl as possible. The reader, in theory, has access to all of this material in the forn several separate manuals for the compil the linker, and the operating system. M Chess relates these various parts of the development environment to each othe ways that most programmers will appreciate.

The book's strength lies in how well its author anticipates the readers' interests

David M. Barnes is a technical writer w came to Oregon Software via Compu-Serve Incorporated, where he develope interests in microcomputers and softw tools.

Calling VMS system service and run-time library routines via Pascal—2

by Thomas E. Hanrahan

VMS system service and run-time library routines perform a wide variety of functions. They control resources available to processes, provide access to process information, permit communication between processes, manipulate character strings, and allow greater flexibility of input and output operations. Many of these procedures are useful programming tools and are available to Pascal programs.

Both the Pascal—2 compiler and support library running under VMS are written primarily in Pascal and rely on calls to VMS routines. One example is the support library procedure p2_get_foreign, used to read input directly from a command line. P2_get_foreign calls the VMS run-time library routine lib$get_foreign to return the contents of the foreign command line that invokes the compiler.

When you call a VMS system service or run-time library routine from a Pascal program, you must supply whatever arguments and calling sequence the utility requires. In all cases, VMS routines follow the standard VAX-11 conventions for calling procedures, so they must be declared as nonpascal when called from a Pascal program. To determine the arguments that a particular VMS routine requires, you must refer to the VAX/VMS manual in which the routine is documented, usually either the *VAX/VMS System Services Reference Manual* or *VAX-11 Run-Time Library Reference Manual*.

In the case of lib$get_foreign, the VMS run-time library routine follows the format:

where:

- *get-str* is a string to receive the text of the command line.
- *user-prompt* is a string to be used as a prompt for additional input if no command-line text is available.
- *rtn-len* is an integer variable passed by reference to return the length of the command line string.
- *force-prompt* is an integer variable set equal to 1 or 0 according to whether you want a prompt issued in all cases or only when command-line text is unavailable.

String arguments are generally passed to VMS routines by reference to VMS "string descriptors." Such string descriptors consist of four components:

- A word containing the length of the string.
- A byte specifying the data type of the argument.
- A byte specifying the class of the argument.
- Four bytes (described by DEC as a "longword") containing the address of the string.

Data-type and class values can be selected from tables provided in the *VAX-11 Run-Time Library User's Guide.*

A VMS string descriptor is defined in Pascal as a packed record whose fields correspond to the descriptor components just described. You begin setting up a string descriptor by defining the string buffer that the descriptor is to reference as some packed array of char. Next, you define two integer-subrange types that fit exactly into a byte (0..255) and a word (0..65535). And finally you define the descriptor itself, as shown in the type declaration of the example on the following page. Notice that pointers generated by the Pascal—2 compiler, such as ^cmdbuffer in the example, are always four bytes in length and may serve as the fourth component of a string descriptor.

VMS system service and run-time library routines may be defined as either procedures or functions, depending on whether or not you want to test for successful completion of the called routine. When declared as a function, the routine returns an integer value. An odd return value indicates successful or non-fatal completion.

In p2_get_foreign, the VMS routine lib$get_foreign is defined as a function with four parameters corresponding to the four required arguments for the routine.

The body of the procedure p2_get_foreign (not shown) is devoted to assigning appropriate values to the arguments that must be passed to lib$get_foreign. For instance, the descriptor txt_descr, references the string that returns the command line text, as follows:

```
with txt_descr do
  begin
  len := h2-l2 + 1;
  class = 1;{ fixed-length string }
  dtype: = 0;
  txt_ref := ref (txt_buf);
  end;
```

The descriptor prompt_descr is similarly assigned values. The string prompt_buf, which prompt_descr describes, is assigned the character string passed to p2_get_foreign from the calling procedure or main program. And the variable force is set to 0 because p2_get_foreign returns a prompt only when command-line text is unavailable.

Once assignments have been made to the various descriptors and variables, lib$get_foreign is called by the statement shown in the example. The routine stores the text taken from the command line in the string txt_buf. The contents of txt_buf can then be copied to the string txt by p2_get_foreign and subsequently passed back to the main program or calling procedure. There, the text can be parsed and processed as needed.

The usefulness of the procedure p2_get_foreign is not limited to passing command line text to the Pascal—2 compiler. As part of the Pascal support library, p2_get_foreign can be called from any Pascal program running under VMS as a foreign command. (See the section entitled "Reading Command Lines" in the recently released Update Package No.1 for Pascal—2 on VMS for further details.)

The ability of Pascal—2 to create an interface such as this one between p2_get_foreign and lib$get_foreign is an important feature of the compiler, allowing you access to the full capabilities of the VAX/VMS programming environment.

```
const
  cmdlinelength = 512;

type
  cmdindex = 1..cmdlinelength;
  cmdbuffer = packed array [cmdindex] of char;
  byte = 0..255;
  word = 0..65535;
  descriptor = ———————————————————————— VMS string descriptor
      packed record
        len: word;
        dtype: byte;
        class: byte;
        txt_ref: ^cmdbuffer; ———————————— pointer filling a longword
      end;

function lib$get_foreign(var line1: descriptor;
                         prompt: descriptor;
                         var length1: word;
                         var forceprompt: integer): integer;
  nonpascal; ———————————————————— creates correct calling sequence

procedure P2_get_foreign(prompt: packed array [l1..h1: integer] of char;
                         var txt: packed array [l2..h2: integer] of char;
                         var length: integer);
  external;


procedure P2_get_foreign;

  var
    ret_status: integer;
    txt_descr: descriptor;
    txt_buf: cmdbuffer;
    txt_length: word;
    prompt_descr: descriptor;
    prompt_buf: cmdbuffer;
    i, k: integer;
    force: integer;

    begin
      :
      ret_status := lib$get_foreign(txt_descr, prompt_descr, txt_length, force);
      :
    end;
```

This code segment from the Pascal support library routine p2_get_foreign *shows in Pascal terms the definition of a VMS string descriptor and a call to the VMS run-time library routine* lib$get_foreign.

Errors, additions to manuals

Roughly every six months, we distribute update packages along with the release of new versions of our software. These update packages are made up of "change pages" documenting changes and additions to be included in each manual. In some cases, the changes describe new features or enhancements implemented in the software. Other times, the emendations are based on information or requests that we have received from readers through Documentation Evaluation Reports (the DER forms at the back of each manual), Trouble Reports, and support calls. (See the list of update packages issued to date.)

The cycle of update packages is staggered and generally depends on a product's original release date. During the interim between updates, we list changes and additions in this newsletter and the Release Notes. Note: whenever a change is made in response to a Trouble Report, we've placed the Trouble Report number in square brackets at the end of the entry.

Oregon Software Update Packages

Update packages are issued at roughly every other release of the software, with changes to intermediate releases being documented by Release Notes and this newsletter. Note: not all products are initially released as version A; Pascal-2 for VMS, for example, began as version 2.1C.

User Manual	Publication Date
Pascal—2 V2.1 for RSX	July 1983
Update Package No.1 V2.1B	October 1983
Update Package No.2 V2.1D	February 1985
Pascal—2 V2.1 for RSTS	August 1983
Update Package No.1 V2.1B	October 1983
Update Package No.2 V2.1D	February 1985
Pascal—2 V2.1 for RT	August 1983
Update Package No.1 V2.1B	October 1983
Update Package No.2 V2.1D	February 1985
Pascal—2 V2.1 for VMS	October 1984
Update Package No.1 V2.1D	May 1985
Pascal—2 V2.1 for UNIX (68000)	August 1983
Update Package No. 1 V2.1D	April 1984

All Pascal—1 manuals

Our April release of the 1.3D Pascal—1 software coincided with our distribution of the 3rd edition of the Pascal—1 User Manual. Like the 1.3D software, the Pascal—1 User Manual has become an unsupported product. Whenever possible, however, we will publish corrections or additions to the manual in the *Pascal Newsletter*, and we encourage Pascal—1 users to send us tips or descriptions of problems that we may pass along to other Newsletter readers.

All Pascal—2 manuals

Page numbers are not provided in this section because they vary from book to book. Instead, the location for changes is described by section name and paragraph or line number where appropriate.

Constants not included in "Dead Code" elimination

Under the description of "Dead Code Elimination" in the section "Compiler Optimization," add the following sentence to the second paragraph:

> The compiler does, however, generate constants in the constants' psect for string literals that appear in write and writeln statements within a dead code region.

[2425]

PASMAT won't format %include files

In the section "PASMAT: A Pascal—2 Formatter," add to the list "Limitations and Errors" the following item:

- PASMAT does not process %include files. Consequently, if the A directive is specified, an identifier is determined by that of its initial occurrence in the file being formatted and not by its form in the %include file.

[2201]

All Pascal—2 manuals for PDP-11 and Professional 300 Operating Systems

Embedded switch limitation

In the "Embedded Switches" section of the "Programmer's Guide," add the following sentence to the beginning of the fifth paragraph:

> You may use up to 25 embedded switches in a compilation unit.

[2760]

Extended-range arithmetic

Replace the first paragraph of the "Extended-Range Arithmetic" section of the "Language Specification," with the following text:

> The normal range of integer variables in Pascal—2 is -32767..32767, but you may also declare variable types in the extended range of 0..65535. A variable with an upper limit greater than 32767 is called an extended-range or "unsigned" variable. Any integer value may be assigned to an unsigned variable and is converted as a bit pattern. If the value being assigned is negative, the error is not trapped at run time, since there is no way for the compiler to tell the difference between a negative "signed" value and an extended-range "unsigned" value. The same sort of implicit transformation is true when an extended-range value is assigned to an integer variable.
>
> The arithmetic operations of addition, subtraction, and multiplication are always signed operations. They treat both signed and unsigned variables as though they were signed. The results for these operations on signed variables are always correct; results produced for unsigned variables are correct except in overflow conditions involving multiplication. Division and modulo are unsigned operations for dividends in the extended range, but they do not treat divisors greater than 32767 as unsigned values. Comparison operations are signed or unsigned according to variable type.

Pascal—2 for RSX

Overlay structures, pages 2-18 and 2-19

Replace the last sentence in the description of the DEBUG2 structure with:

> For programs using single-precision, you have to specify SING, a special variant of the SINGLE co-tree, designed for use with DEBUG2. SING accounts for the fact that the Debugger is itself a Pascal program compiled with double precision.

In the third overlay example on page 2-19, change the SINGLE co-tree to read SING. [2803]

Stack overflows, page 2-31

Add the following text immediately before the section entitled "The 'Space' Function."

Caution

> Very large stack overflows, those that extend beyond the heap and into the program code sections of memory, have the potential for causing serious problems. In some cases it is possible for the error to prevent the appropriate error message from being printed or the program from properly terminating. In some rare instances, the condition can even lead to the disruption of the computer's operating system. It is the programmer's responsibility to avoid such excessively large overflows by controlling the size of local variables and value parameters passed to procedures.

Error walkback with I & D space programs, page 2-42

At the the bottom of the page, insert the following paragraph:

> Error walkback cannot be used with Instruction & Data (I & D) space programs available on RSX-11M/PLUS. Normally (in non-I & D space programs), the walkback routine examines instructions to find special markers and pointers to data areas, which contain procedure names and statement locations. But, in I & D space programs, instructions and data are kept separate, and this, coupled with the fact that data sections can be overlaid, makes it difficult for the walkback routine to function properly. In some I & D space programs, the walkback routine reports addresses instead of procedure names, but in other instances, the walkback procedure can abort with an odd address error or memory management trap. You should generally compile all I & D space programs with the /nowalkback switch to avoid such problems.

[2516]

Program example, page 2-59

In the program example, change th statement:

```
Efn := 1; {use event flag}
```

to read:

```
Efn := (16*256) or 1
```

The new line checks for all 8 words in the Status Block. [2800]

Debugging with I & D space programs, page 4-1

At the the bottom of the page, insert the following text:

Caution

> The Debugger cannot be used on I & D space programs. The separation of instructions and data generated by such programs makes i difficult for the Debugger to properl trace procedures and statements. (Se "Run-Time Error Reporting" for more details.)

Pascal—2 for RT

Command line error, page 1-3

The command line for running the program CUSTOM is wrong. It should be changed to:

```
.R CUSTOM
```

Foreground Operations, page 2-49

The last sentence on the page should be changed to read:

> The period after 1024 signifies a decimal value.

Pascal—2 for UNIX (68000)

A number of pages in the manual contain minor errors that can be best changed by penciling in the changes as described:

Multiple embedded options, page 2-5

Below the sample embedded directive lines:

```
{$noindex,norange}
{$noindex}{$norange}
```

Add the sentence: "A space between options in the first command line, i.e. {$noindex, norange}, is not allowed and the compiler treats everything after the space as an ordinary comment. "

Selective debugging, pages 2-5 and 2-6

In several places on these pages, the manual states incorrectly that the debugging option can be turned "on" or "off" for sections within a module. The correct explanation is that the embedded options $debug and $nodebug actually have no effect on the current version of the Debugger for UNIX/68000. Debugging can occur only for an entire compilation unit and may be invoked only by specifying the -debug option on the compilation command line.

Removal of parameters from the stack, page 2-15

The last sentence in the fourth paragraph should read, "It is the responsibility of the calling procedure to remove parameters from the stack."

Storage allocation for packed record, page 2-17

The last sentence in the paragraph should read, "beginning at bit 8 (most significant bit)."

'Read' and 'Readln' procedures, page 3-15

Replace the existing heading with 'Read' and 'Readln' Procedures and add the following text after the section's first paragraph:

> For variables of type subrange, read and readln statements accept values for the variable outside the limits defined for the subrange and the error is not recognized until such values are actually used. Error checking at the time of input, for values outside the limits of a subrange, is the programmer's responsibility.

Program listing, page 3-21

The type declaration of Word is incorrect in the program listing. The correct limits should be defined as 0..4294967295 and the program produces different results from the ones shown in the example.

Process termination values and statements, page 4-2

Add the following paragraph just before the final paragraph on the page:

> A process termination (exit) value and statement is given by the Debugger each time a program terminates, whether termination is normal or the result of some error. Exit values are either 1 or 0. An exit value of 1 always indicates that termination is the result of a run-time error and is preceded by the appropriate run-time error message. An exit value of 0 indicates either normal termination or termination because of some error other than run-time, such as a bus error. Except for normal terminations, error messages are also given for terminations that yield exit values of 0.

Passing parameters in the Debugger via Go, page 4-10

Replace the existing description for the G: Go command with the following:

> The G (Go) command begins executing the program at MAIN,1; this command may be used at any point to restart the program. See the example after the C command.
>
> If you debug a program that requires files to be passed as arguments, such files can be passed as parameters (similar to those in a write statement) with the G(*'file'*) command. For example, suppose that the program test.pas requires that data.dat be passed at some point during program execution. Compile test.pas and invoke the Debugger with:
>
> pc -debug -output test test.pas
> pdb test
> }
> Use the G command to pass data.dat by entering:
>
> } g('data.dat')

Note that the argument is enclosed in single quotes (' '). Multiple arguments must be enclosed individually in single quotes and separated by commas, as:

} g('arg1','arg2','arg3',...)

'For' statement index entry is incorrect, page Index-2

The correct page reference for for statements is 3-28.

Pascal—2 for VERSAdos

The manual, soon to be released as a new edition, is being thoroughly reformatted to conform to the style of our other Pascal—2 manuals. The text contains minor corrections and additions throughout. Appendices listing compilation and run-time errors will be brought up to date.

Pascal—2 Cross-Development System

An update package is being prepared that reflects the switch from the OASYS cross-linker/assembler to the Oregon Linker/Assembler. The update package will also include an index for the first time.

Continued from Page 1

and several cross-compiler configurations. We will provide more details later.

Other developments

Maintenance work is continuing on our other products. The latest update for our VAX/VMS native compiler, version 2.1D, was released in early June.

Technical problems in the first shipments of version 2.1D of Pascal—2 for the PDP-11 caused us to pull back that release. The problem has been corrected, and customers who received the faulty shipments have been updated. All shipments of Pascal—2 for RSTS, RSX, and RT-11 are now Pascal V2.1D. Work on 2.1E will begin this summer.

The Log: errors, work-arounds, changes

This log describes significant changes in Oregon Software products. As a service to our users who have reported problems to us in the past, we have provided Trouble Report numbers where possible. At the end of this section, we've also listed reported bugs that have been verified by us as being problems with the software. Where possible, work-arounds for know problems are provided after the description of the error.

All Pascal—1 compilers

The following is a list of problems corrected by the 1.3D release.

Number	Description
TR 508 TR 1213	The assignment of a value to a function identifier went undetected. Now, the compiler generates the error message Illegal assignment.
TR 845 TR 1357	The compiler failed to detect an illegal expression containing two consecutive operators. Now, it issues the error message Bad expression.
TR 897a	Pascal—1 incorrectly evaluated boolean operations on integers with magnitudes approaching maxint. Such expressions are now correctly evaluated.
TR 899 TR 1262 TR 1993	External procedures declared twice in the same program caused the compiler to crash. Now, the error message Bad procedure name is issued.
TR 1149	Expressions involving combinations of pred, ord, and succ functions occasionally returned incorrect results. Such expressions are now correctly evaluated.
Unassigned	The get procedure no longer fails when accessing a file of type text.

Pascal—2 for VMS

The following is a list of bugs fixed in version 2.1D.

Number	Description
TR 2513	The Pascal—2 command line prompt displayed two different strings. A single carriage return generated the string PAS > ; a second carriage return resulted in MP2 > . The string PAS > is now displayed in all cases.
TR 2826	The installation command file, INSTALL.COM, failed to create the Pascal—2 help library when an existing library structure was not already in place. Documentation has been added to the Installation Guide describing steps to take in such instances.
Unassigned	The I/O control switch /temp failed to deallocate space used by temporary files, even though it deleted the corresponding file directory entries. The /temp switch now correctly deletes temporary files.

Set problems corrected

TR 2501 TR 2511	Defining a set as a structured constant caused the compiler to abort with an access violation. The compiler now generates the correct syntax-error messages for such conditions.

TR 2527
TR 2548

Certain set instructions failed to produce the correct code and in some cases resulted in an access violation.

TR 2549

The compiler failed to generate an error message when a variable was declared to be a set of an enumerated type consisting of more than 256 elements. Such set declarations now produce the compilation error message Set types must have base in the range 0..255.

TR 2759

Set expressions with non-scalar elements, such as strings, caused the compiler to crash.

Conformant array bugs fixed

TR 2465

Certain programs hung when a call was made from within a for loop to a procedure with a conformant array parameter. Such programs now execute correctly.

TR 2526

Undefined variables passed to conformant array parameters of a procedure or function were not trapped by the compiler and resulted in access violations. Such variables are now properly labeled by the compiler as undefined identifiers.

TR 2574
TR 2602

Passing a structured constant by reference to a conformant array parameter resulted in the error message Too many nodes in procedure. The compiler now generates the correct compilation error Variable name expected.

Readin', writin', and . . .

TR 2555

Integers written to a file of real were not converted to real number format. This led to a run-time error when an attempt was later made to read such files. The compiler now generates code that converts integers to reals when they are written to a file of real.

TR 2713b

Attempts to read the real number 0.0 caused programs to loop indefinitely. The compiler now generates code that correctly reads the value 0.0.

. . . 'Rithmetic problems fixed

TR 2651
TR 2720

Incorrect values were returned for mod operations performed on negative numbers. Mod now generates results according to the Pascal standard.

TR 2815

The absolute value function, abs, incorrectly used a source register for a destination register. As a result, a variable with a negative value was changed to positive when it was used as an argument of abs. The compiler now processes absolute values correctly.

Unassigned

The mod function returned incorrect results for unsigned integers when both the dividend and divisor were greater than 2147483647.

'Reset' and 'rewrite' bugs fixed

TR 2576

Certain attempts to reset the standard file input caused a run-time error. The statement reset(input) now performs correctly.

TR 2616

Text was stored in a file improperly when an attempt was made to rewrite the file after it had already been opened for writing. The compiler now generates code that correctly resets a pointer to the file's buffer.

Debugger problems corrected

TR 2684

When a command line included both file extensions for output files and the /debug switch, as shown in the example:

```
$ pas2 main.obj,main.lis = main.pas/debug
```

the VAX Linker detected illegal record sequences in the resulting object file. The compiler now generates the correct object code in these cases.

Unassigned	Variables stored in registers by the compiler returned incorrect values when watched or written by the Debugger. The Debugger now handles variables stored in registers correctly.
Unassigned	Variables in the last line of a program were not previously accessible to the Debugger. Such variables are now available.
Unassigned	The "watch" Debugger command W could not be set before the first line of a program was executed. With this release, watched variables can be specified before you begin execution of a program.

Pascal—2 for UNIX (68000)

Version 2.1G corrects the following problems with the compiler.

Number	Description
TR 2291	The system call getpid, a C function declared as nonpascal in a Pascal program and used to return a program's process ID number, caused the run-time error Illegal instruction (core dumped) whenever it was called from within a write or writeln statement. The system call now functions properly within write and writeln statements.
TR 2666	Mod operations computed incorrect results when performed on variables passed as parameters to a procedure or function. Such operations now generate correct values.
TR 2667	The compiler interpreted file names to be all lower case letters when searching for %include files. As a result, lookup failed on %include files whose names contained upper cases letters, such as Test.pas. The compiler is now case sensitive in searches for %include file names and recognizes file names with upper case letters.

Problems with arrays corrected

TR 1917

The compiler crashed whenever a variable of type subrange was used as the bound in an array declaration, as with the variable z in the example:

```
type
  a = 1..7;
  x = 1..7;
  y = 1..7;

var
  z: a;
  ex: array [x,y,z] of integer;
```

The compiler now generates the correct compilation error message Bad constant in such cases.

TR 1961
TR 2172
TR 2531

The run-time error message Integer overflow resulted from illegal type declarations of the form type T = array [1..const-1], where an expression rather than a constant was used to define the limit of an array. The error is now trapped during compilation and identified with the proper syntax error message: ',' expected.

TR 2660

Certain Subscript out of bounds errors were incorrectly identified as variable subrange exceeded errors. The problem occurred when a variable of type subrange was used as the index of a packed array and a value was assigned to the variable that exceeded the bounds of the index but not the limits of the subrange. The correct Subscript out of bounds error message is now given in such cases.

TR 2679

Changing the value of a single element in a packed array that was a field of a packed record caused other elements of the array to be similarly modified. Such operations on packed arrays within a packed record are now performed properly.

Known Bugs

We haven't completed our work on all the Trouble Reports that we have received; the following is a description of those problems we've verified. Except for the problems noted with the now frozen Pascal—1 compiler, the bugs listed here are all scheduled for correction in subsequent releases.

Pascal—1

The following is a list of bugs outstanding in 1.3D.

Number	Description
TR 465	When calling FORTRAN library functions using the `fortran` procedure option and the `/X` compilation switch (double precision reals), the function return value is lost.
TR 846	Double-precision function calls occasionally leave the stack in an inconsistent state.
TR 968	IMP-processed code generates a `Branch out of bounds` error during assembly.
TR 1069	For the RSTS compiler, the `reset` procedure uses an assigned account number as the default directory.
TR 1348	Assignment from a function passed as a parameter to a procedure generates an `Incompatible type` error.
TR 1552	A recursive procedure with a procedure parameter does not compile.
TR 1811	The compiler fails to detect the second occurrence of an illegal operand.
TR 2174	The compiler occasionally fails to diagnose a procedure call containing an incorrect number of parameters.
TR 2267	Some file handling operations (`Close`, `Reset`, `Rewrite`) may generate spurious code unless followed by a semicolon at their call sites.
TR 2330	In a program with nested procedures, a `goto` from within a `for` loop in the innermost procedure occasionally fails to find its destination in the outer procedure.
TR 2335	When writing to a file from a subscripted variable, the subscript calculation is sometimes incorrect.
TR 2388	When a variable of type `text` is assigned to another variable of type `text` the compiler occasionally fails with an odd address trap instead of issuing an error message.

Pascal—2 for VMS

The following are known problems in version 2.1D.

Number	Description
TR 2514b	The compiler fails to return a file name when it reports a run-time I/O error.
TR 2550	In certain cases, run-time errors that should generate the message `Variable subrange exceeded` are flagged as `Array index out of bounds`.
TR 2610	When specified on the compilation command line, an illegal file name with two consecutive "dots" (i.e., TEST..PAS), causes the compiler to generate the run-time error message `Can't open file`.
TR 2473 TR 2514d	The compiler fails to terminate with a "severe error" status when it detects compilation errors.

TR 2642	In certain cases the compiler fails to recognize an Array index out of bounds condition when compiled with checking enabled. Compiling with the /nocheck switch causes the errors to be correctly identified.
TR 2713a	The function trunc does not check for extended-range values and converts all unsigned real numbers to signed integers.
TR 2744 TR 2837	In certain instances the compiler generates incorrect code for empty case statements with constant case selectors.
TR 2768	The Dynamic String Package procedure insert( ) fails to truncate the target string properly when *max-len* characters is exceeded. Such overflows result in the run-time error Array index out of bounds.
TR 2776	The run-time error message File not open is produced when you use a single readln statement to input a packed array of characters and an integer from a text file, as in: readln(text, str, i); where text is a file of text, str is a packed array of characters, and i is an integer.
TR 2829	Write statements generate spurious carriage returns when escape sequences or VMS run-time library calls are used to format screen output.
TR 2862	Readln(i), where i is of type integer accepts all types of input. As a work-around, use read(i) followed by readln to input integers.
TR 2863	Lack of an argument for the support library routine iostatus generates the error message Too many nodes in the procedure rather than the correct error message pair '(' expected and ')' expected.
TR 2870	The compiler incorrectly parses packed set elements of packed records when the set contains more than 32 members.
Unassigned	The Debugger prompt } is followed by a spurious carriage return, causing your input to be written one line below the level of the prompt. Consequently, a typical Debugger command appears on your screen as: } B(MAIN,5)

Pascal—2 for UNIX(68000)

The following are known problems in version 2.1G.

Number	Description
TR 1961 TR 2241	Incorrect code is generated with certain element assignments of packed arrays of packed records when the record fields involved are sets. Pascal—2's optimizations cause the resulting code to be generated with byte rather than word instructions and lead to incorrect results.
TR 2536	Certain programs that attempt to write elements of arrays that are themselves packed arrays of char return the run-time error message Illegal instruction (core dumped).
TR 2586	A mismatch between a type definition and a structured constant definition using that type causes the compiler to crash and return the error message Illegal instruction (core dumped).

TR 2589

Invoking both of the compiler options -debug and $nolist during a single compilation causes the compiler to generate an incomplete listing file and the Debugger crashes.

TR 2730

Specifying a pointer to a file variable in the argument of a rewrite( ) statement causes a core dump upon termination of the program. The program compiles and runs correctly except for the termination error, and the rewrite( ) statement is properly executed. The core dumped error does not occur when a file variable is passed directly to the rewrite( ) statement. This error was first reported in Version 2.1E.

roblems in the Debugger

TR 2445

The Debugger does not have access to function values because the symbol table file currently only recognizes variable names.

TR 2658a

The Debugger does not write correct values of constants.

TR 2658b

The Debugger fails to return an error message when you to attempt to assign a value to a constant. The assignment, however, is never made, and the value of the constant remains unchanged.

TR 2731

When a run-time error is encountered in a program being debugged, the Debugger is not able to recover the stack frame from which the error occurred and, as a result, it loses access to the program's variables. This condition will be corrected by the introduction of a post-mortem analyzer in a future release. As a temporary work-around, when you encounter a run-time error during debugging, set a breakpoint at the statement in which the error occurs, then restart the program and run it to that breakpoint. Once you reach the statement, you can accurately probe variables immediately before the point at which the run-time error appears.

TR 2732

In modules compiled with the $own option, Debugger commands taking an argument true, false, or nil are flagged as Unknown identifier errors when set within a stack-frame context.

As a work-around, return to the main program, set the command you wish to use, and proceed back to the breakpoint you want to examine. You can accomplish this using the command line:

```
} e(1);H(true)
```

Oregon Linker/Assembler

The Oregon Linker/Assembler has two outstanding bugs at this time.

Number	Description
Unassigned	The linker map fails to display sections containing no code. Sections that only define storage declarations do not appear on the map.
Unassigned	Extraneous spaces, special symbols such as $, and version numbers on the command line are flagged as illegal.

Newsletter Index

This index catalogs topics that have appeared in the *Pascal Newsletter.* Entries are listed by newsletter number (in bold) and page number. Readers' comments on the format as well as the contents of this index are welcome.

K, L

M

N

O

P, Q

R

S

T

U, V, X

Pascal
NEWSLETTER
OREGON SOFTWARE

6915 S.W. Macadam Avenue
Portland, Oregon 97219